OPTIMIZING BARNES-HUT SIMULATIONS FOR MANY-CORE SUPER COMPUTERS USING THE SCALABLE PARALLEL RUNTIME

BY

KURTIS LEE NUSBAUM

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Marc Snir

ABSTRACT

It is now generally accepted that the road to Exascale Super Computing will no doubt include increasing core counts on individual compute nodes. The Scalable Parallel Runtime (SPR) is a project being developed at Sandia National Laboratories which attempts to address the challenge of efficiently utilizing the power of these new many-core systems. SPR is a combination of the Qthreads Library, Portals, and MPI. In this paper, we investigate optimizing the Barnes-Hut simulation by using the PPL Runtime which we implement using SPR. PPL provides a PGAS runtime with one-sided communication facilities, ideal for irregular applications with dynamic communication patterns like that of Barnes-Hut.

For my mother who has always loved me unconditionally,
and for my father who gave me everything he had to give.

ACKNOWLEDGMENTS

I would like to thank:

- Marc Snir and the XGC project, without whom this research could not have been done
- Kyle Wheeler and Dylan Stark of Sandia National laboratories, for creating and maintaining SPR
- Junchao Zhang of the University of Illinois, for his amazing debugging help
- The ACM at UIUC, whose random assortment of knowledge proved invaluable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SPR	Scalable Parallel Runtime
FEB	Full-Empty Bit
PGAS	Partitioned Global Address Space
PPL	Parallel Programming Language
RDMA	Remote Direct Memory Access

CHAPTER 1

INTRODUCTION

As the field of High Performance Computing looks to increase computing power another order of magnitude from petascale to exascale, new solutions are required to the old problems of parallelism and communication. It is widely accepted that future super computers will see, along with an increase in node count, an increase in per-node core count. In order to properly utilize this new level of parallelism, new abstractions must be used. Simply assigning one MPI rank to each core is a solution that does not work in many cases. A more intelligent runtime will be required to actualize the full potential of future hardware. The Scalable Parallel Runtime (SPR) currently being developed at Sandia National Laboratories looks to address the problem of many-core parallelism. By utilizing light-weight threads with small stacks, the runtime can rapidly execute millions of software-based threads.

The increase in computing power is not without purpose. High-performance physics applications are always looking for ways to make their simulations bigger while achieving higher resolution. One such physics simulation is the Barnes-Hut n-body simulation [1]. This novel physics simulation, which runs in $O(n \log n)$ as opposed to $O(n^2)$ for a naïve n-body simulation, presents unique issues for High Performance Computing. It's communication pattern is irregular and unpredictable, with many small messages needing to be sent at each iteration.

Problems that require irregular, unpredictable communication, such as Barnes-Hut, are hard to express using message passing. They are much more suited for being expressed using RDMA operations in a PGAS environment. However, naïve implementations of PGAS languages, such as UPC [2], achieve bad performance. Zhange et. al. [3] have developed a new PGAS runtime with RDMA, named PPL, that can be used to achieve good performance for problems like Barnes-Hut.

In this paper, we experiment to see if SPR can be used to efficiently support

the constructs of PPL. Our experiments show that, while it's many-core parallelism facilities perform well, the multi-process communication portions of a PPL implementation based on SPR do not perform for applications like Barnes-Hut. Namely, the RDMA operations provided by SPR perform several orders of magnitude too slow in order to be used for applications like Barnes-Hut simulations. Future work is needed to determine if the problem is in the design of SPR or our use of SPR.

CHAPTER 2

BACKGROUND

2.1 Barnes-Hut

The Barnes-Hut algorithm is an n-body simulation algorithm[1] that was introduced in 1986 by Josh Barnes and Piet Hut. Previous approaches to n-body simulations either ran each time step in $O(n^2)$ time or produced loss of accuracy and generality while requiring specialized code for the simulation at hand. The algorithm presented by Barnes and Hut was novel in that each time step now only ran in $O(n * log(n))$ time, loss of accuracy was controllable, and no special code was needed for particular simulations.

At a high level, the Barnes-Hut algorithm works as follows for a three-dimensional simulation:

1. All bodies are divided into groups and stored in an Octree, with each tree node representing a region of space. The root node represents the smallest rectangular box containing all the bodies. The root node is then divided into eight sub-regions which make up the child nodes of the root node. This division continues until each node contains either one or zero bodies. The resulting Octree thus contains two types of nodes: internal nodes and external nodes. Internal nodes have children while external nodes do not.

2. Once the Octree has been built, the force on each body is calculated. To calculate the force on a given body b, the other nodes in the Octree are examined. When examining an internal node i whose center of mass is far enough away from b, all the bodies in that region of space are simply

[1] By n-body simulation, we mean a simulation of the movements of n bodies over a period of time, with each body exerting forces on one another. The simulation is divided into time steps and the movement of each body is calculated at each time step. One example of an n-body simulation is that of two galaxies colliding.

treated as a single body. If the internal node is not sufficiently far away, it is "opened" and it's children are examined. If a child only contains a single body b', the force exerted by b' is added to the calculation of the total force being exerted on b. If the child node contains no body, it is simply not examined.

Whether or not i is far enough away is determined by the quotient $\theta = \frac{w}{d}$, where w is the width of i's region and d is the distance between b and the center of i. b is considered to be far enough away when θ is below some specified threshold.

A basic parallel implementation of the algorithm consists of doing the following phases at at each time step:

1. Build Octree
2. Update Cells
3. Partition Octree
4. Compute Forces
5. Advance Bodies

The above phases (which we describe in detail in Section 2.3.2) are repeated at each time step in the simulation, with the majority of time being spent in the force calculation phase. For large values of n, the result is that each time step is executed in $O(n * log(n))$ time instead of $O(n^2)$. Furthermore, the accuracy of the simulation can be controlled by adjusting[2] the value of θ.

2.2 Scalable Parallel Runtime

The Scalable Parallel Runtime is a project currently being developed at Sandia National Laboratories that consists of three main components: the Qthreads Library, Portals, and MPI [4, 5, 6]. The Qthreads Library provides all of the many-core parallelism facilities for SPR while portals and MPI

[2]Note that when $\theta = 0$ the algorithm degrades into a naïve, $O(n^2)$ version of the algorithm [1].

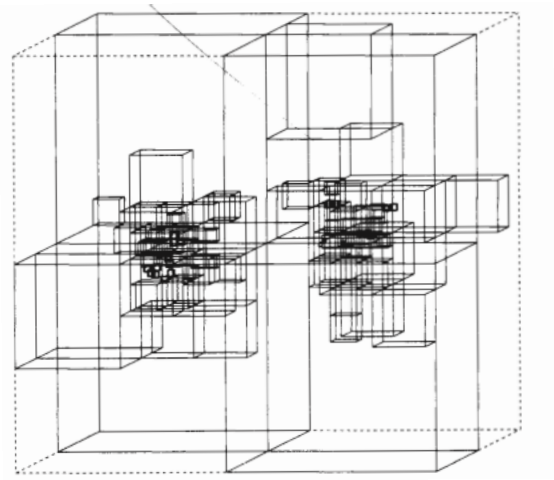

Figure 2.1: An example of a region of space that has been divided up into Octree sections.

provide communication abilities. Portals is used for small RDMA operations while MPI is used for collective operations. The typical use case of SPR involves using the Qthreads Library to achieve intra-process parallelism while using Portals and MPI to achieve inter-process parallelism.

2.2.1 Many-core Parallelism

The Qthreads Library itself offers a unique and portable API that, while similar to the well known Pthreads API [7], allows for the creation of massive numbers of threads across a vast array of different hardware platforms [4]. Threads are implemented as tasks that are given small stacks (only 4kb by default) which can be context switched rapidly in software. The key synchronization component of the Qthreads Library is full/empty bits. Each variable of type `aligned_t` is marked with a special bit that is either in a "full" or "empty" state. Programs written using the Qthreads Library can wait on this state, i.e. a thread will not progress until a certain piece of memory is marked as being full. Furthermore, waiting on full/empty bits (FEBs) is efficient. The runtime knows exactly when each FEB is set to full or empty and will wake up appropriate threads accordingly, i.e. threads do not have to poll the state of FEBs. It is possible to implement full/empty bits in hardware on machines such as the Cray XMT, although in most cases FEBs are implemented in software.

Since threads are so light weight, it is perfectly reasonable to spawn massive amounts of them. Tests have shown the Qthreads Library can be used to create and execute one million threads in less than a minute [4]. With the ability to spawn threads easily, the Qthreads Library can be used to create things like parallel recursive programs. Each recursive function call is simply another thread that is launched. This allows for greater overlap of both work and communication between individual threads. The spawner of these threads can then simply wait on a FEB.

The scheduling of threads is handled by a custom scheduler in the Qthreads Library itself. The scheduler uses a small group of hardware threads to manage the large collections of qthreads. Each group of qthreads is known as a shepherd and also represents a region of locality. Shepherds can be manually configured by the user (i.e. one shepherd per socket) or intelligently configured by the runtime itself using hardware introspection, e.g. hwloc [8]. Memory allocated by qthreads belonging to the same shepherd is guaranteed[3] to be in the same NUMA region as other qthreads in the same shepherd.

When a qthread in a shepherd blocks on a FEB, the shepherd can immediately context switch to an alternate qthread that has work to do. The shepherd will then never context switch back to the blocked qthread until the FEB is set to full. This is particularly useful for network operations. When blocking on the completion of a one-sided communication operation, the blocked thread will not come back until on or after the communication operation has finished.

2.2.2 Multi-Process Parallelism

SPR provides RDMA via the Portals library. In addition to traditional PUTs and GETs, SPR also provides the ability to fork threads on remote processes. This allows arbitrary actions to be preformed on remote data. Instead of moving data to where computation is being done, computation can now be moved to the data. However currently, forking threads on remote processes is very expensive.

SPR provides collective operations via MPI. Currently, the only collective operation exposed by SPR is a collective barrier. That said, application

[3]There are some instances where memory allocated by qthreads in same shepherd won't be allocated in the same NUMA region, but these instances are few and far between.

developers may make their own calls to any MPI function safely. The process ranks used by SPR are the same as those used by the MPI runtime, so making calls to arbitrary MPI functions is trivial.

2.3 PPL and Barnes-Hut

2.3.1 PPL

In 2011, Zhang et. al [3] developed a new implementation of the Barnes-Hut algorithm using UPC [2]. Their work showed considerable performance increase over the default UPC implementation of the Barnes-Hut algorithm which was itself almost a direct copy of the SPLASH-2 benchmark in [9]. Furthermore, they abstracted out commonly used patterns into a library which they named PPL. PPL is a C++ PGAS library which provides three main constructs: Global Pointers, Global Vectors, and Global Variables.

A Global Pointer can point to memory located on any process within the system in which the program is running. Dereferencing a Global Pointer retrieves the pointed at information, even if that information is located on a remote process. In the UPC version of PPL, Global Pointers were easily implemented as a simple wrapper around the Global Pointers which UPC provides natively. Additionally convenience functions were also added. Global Pointers can be created by explicitly allocating memory on a given process, or be created as a reference to data located in a Global Vector.

Global Vectors are one-dimensional co-vectors, meaning each process has it's own copy of the vector. They are similar in many ways to the co-arrays provide by Fortran [10]. However, Global Vectors can shrink and expand in size dynamically (although these operations are quite expensive). Global Vectors also allow the programmer to easily index into a remote process's copy of the vector and perform collective operations on the set of vectors as a whole (e.g. a global reduction).

Global Variables are variables that can be referenced by all processes, but have their value explicitly cached. They act just like normal variables except all changes made to the variable's value are also made to the version of the variable that is located on the root process. However, when changing the value of global variable, the changed value is not propagated to other

processes automatically. In order to retrieve the most up-to-date value of the variable, a process must explicitly request to cache the current value of the variable. Global Variables are a simple method of propagating a single value out to many process only when the processes absolutely need the most up-to-date copy of the variable.

PPL also provides a number of other convenience functions such as remote atomic increments, blocking and non-blocking RDMA functions, collective barriers, and process information queries.

2.3.2 Barnes-Hut Algorithm

The Barnes-Hut algorithm implemented by Zhang et. al [3] shares the same overall parallel algorithm as the UPC and SPLASH-2 code [9, 11, 12]. The algorithm executes these five phases at each time-step:

Build Octree Each process inserts the bodies it currently owns into a shared tree structure. The process also keeps track of the cells it creates while inserting bodies as those cells are now also "owned" by the process.

Update Cells The center of mass and cost[4] for each cell are computed via a bottom up pass through the tree. The cost for a cell is equal to the sum of the costs of all it's children. The cost and center of mass computation are done by the process which owns that cell.

Partition Octree The tree is split up into p segments of consecutive leafs where each segment has approximately the same total cost and p is the number of processes involved in the computation. The ith process is assigned the ith segment. Values for each tree node in the ith segment are then migrated to process i, as process i now owns them.

Compute Forces The cost and force on each body is computed by the body's owner. The simulation spends the majority of it's compute time in this phase of calculation.

[4]We use the term cost as an estimate for amount of work that needs to be done for the force calculation of a given body. We define the cost of a body as the number of interactions it has with each cell in the Octree. Bodies that interact with more cells will require more computations, therefore we say they have a higher cost.

Advance Bodies Each body has it's new acceleration, velocity, and position calculated. Each process then computes the boundaries of the box containing its bodies and the new boundaries for the root cell are computed via global reductions.

This algorithm works well because it addresses the two main issues with implementing Barnes-Hut in parallel: load-imbalance and locality. Load imbalance occurs due to the fact that the amount of computing done on a given body differs from body to body, i.e. a particular body might be close to many others and need to open more internal nodes than a different body. The Partition Octree step of the algorithm addresses this. Since the computational cost of each body is tracked, the Octree can be partitioned in such away that all processes will have approximately equal computational work to do for any given time step.

Locality becomes a concern when a given body needs to open internal nodes that are located on remote processes. Therefore it is better to locate bodies that are near one another on the same physical process, as they will most likely need to access the same information (including information about each other). Once again, this concern is addressed in the partitioning phase of the algorithm. Only sets of contiguous nodes are doled out to each processor, thus ensuring that at least a significant portion of bodies needed for the calculations within a particular group of bodies are relatively close to one another in terms of data location.

CHAPTER 3

EXPERIMENT

3.1 Implementation

Our implementation of Barnes-Hut utilizes a new version of the PPL runtime[1] designed by Zhang et. al. [3]. Essentially, our implementation of PPL involves substituting the use of UPC facilities in favor of the SPR facilities. The following describes how we implemented various parts of PPL using SPR as well as changes to the force calculations we made in order to further increase communication and computation overlap.

3.1.1 PPL

Global Pointers

Global Pointers are implemented using a templated class which is essentially a simple wrapper around two pieces of information: the process on which the data is actually located, and the address where the data is located on that process. The template type corresponds to the data type of the variable to which the pointer is pointing. If a Global Pointer g is located on process n then dereferencing g is as simple as dereferencing a normal pointer on process n. If the Global Pointer g' is not located on process n then the Global Pointer class will use the RDMA facilities provided via SPR to retrieve the pointed at data.

Pointer arithmetic is accomplished by simply modifying the value of the memory address. This allows for Global Vectors to return Global Pointers to elements in a vector while letting the application developer use the Global

[1] A full outline of the PPL Specification as implemented by the program described in this paper can be found in Appendix A

Pointers like they would a pointer to an element in a regular array.

The default constructor of a Global Pointer simply returns a null version of the Global Pointer class. In order to actually allocate a Global Pointer, the application developer must use the `ppl_alloc` function. This function allocates space for the pointer on the process which makes the call to `ppl_alloc` and returns an appropriately constructed Global Pointer. The application developer must also explicitly call `ppl_free` when they are done with the pointer. This will free the memory that was allocated in `ppl_alloc` and set the Global Pointer to null.

Global Vectors

Global Vectors are implemented by using a templated class. The template parameter is the data type of the data contained in the Global Vector. The class has two main attributes: an array containing the values of the processes's vector (the value vector), and an array of pointers to each other process's value vector (the world element pointers). The world element pointers array is the same for every process. The pointers in this world element pointers array are specific to the memory address space for their specific process. For example, the third element in the world element pointers array would be a pointer in the address space of the third process. This pointer would point to the first element of the value vector of the third process. This array of pointers is created by doing a global all-gather. Each process shares the pointer to the first entry in it's value vector with every other process.

Access to values in a processes's copy of the value vector are done in the same way as regular array element access. Access to values on another process's value vector are done via the RDMA operations provided by SPR. Values are resolved by using the target process's pointer found in the world element pointer array. For example, if node n wants the third value in the value vector of process $n+1$, it would take the address in the world element pointers array at index $n+1$, add three to that address, and then use an RDMA GET to retrieve the value using the computed address.

All-reduces for Global Vectors are implemented fairly inefficiently at this time. All value vectors are copied to the root process, and then the reduce operation is applied to them there. Once all the vectors have been copied and reduced, the resulting value vector is then broadcast to all other nodes. This

implementation is due to the need to support arbitrary reduction operations. MPI requires all user defined reduction operations to be preregistered using `MPI_Op_create`. We are currently not ready to introduce the abstractions needed to accommodate for this.

One other implementation detail worth noting is that several operations require a re-exchange of world element pointers array. These operations include renew, resize, and the copy constructor. The copy constructor results in completely new, duplicate vector being created, with it's own value array and world element pointers.

Global Variables

Global Variables are implemented with a very simple templated class. The template parameter specifies the type of the data being stored in the Global Variable. The Global Variable has two main attributes: the actual value being stored, and a pointer to the value on the root node. This pointer is how non-root nodes access the current value of the Global Variable. Setting the value of the Global Variable is as simple as doing an RDMA PUT to the root address and caching the variable's value is as simple as doing a RDMA GET from the root address. The root address is shared with all other nodes at construction time via an MPI broadcast.

Other Functions

Several other facilities are provided by PPL:

- Batched synchronous and asynchronous RDMA operations are achieved via simply looping over an array of Global Pointers and calling the appropriate RDMA operations via SPR.
- Atomic remote increments are accomplished using SPR's remote thread forking facility. A remote thread is forked on the process where the atomic increment is needed, the atomic increment is performed, and then result is delivered back to the forking thread.
- Environment information regarding the number of processes and calling process id are achieved through simple wrappings of calls to `spr_num_locales` and `spr_locale_id`.

- An initial call `ppl_init` is required in order to setup the runtime as well as a final call to `ppl_fini` to tear down appropriate facilities.

3.1.2 Recursive Parallelization of Force Calculations

The original algorithm used for calculating the force and cost of each body as implemented by Zhang et. al [3] looped over the owned bodies on a particular process in serial. Since walking of the Octree is required for each body, serial recursion is also preformed during this step. Parallelism was achieved through using many processes to work on different portions of the Octree.

In addition to allocating different portions of the tree to multiple processes, we sought to achieve additional communication/work overlap using some of the facilities provided by SPR. First, we modified the loop over owned bodies to be executed in parallel using the `qt_loop` function[2] which splits up the array of owned bodies and assigns sections of the array to a group of new threads that are spawned. This allows a single process to be making progress on multiple bodies at the same time. And if the force calculation for a body is blocked on an RDMA request, other bodies can still make progress.

Second, we took advantage of Qthread's light-weight threading model to fork a new thread at each recursive step in the walk of the tree for each body. This allows us to further overlap the computations with the communication that is also taking place at the same time. If any given thread is blocked waiting for data about a body that is located on a remote process, there will most likely still be progress that can be made for the force calculation of that body in one of the other portions of the tree.

3.2 Tests And Results

We conducted a series of tests on the Taub computing cluster located at the University of Illinois at Urbana-Champaign [14]. Our tests consisted of running our implementation of the Barnes-Hut algorithm while varying the number of processes used and the number of bodies involved in the calcu-

[2]The semantics of `qt_loop` are quite similar to those of the `parallel_for` found in Intel's Threading Building Blocks [13].

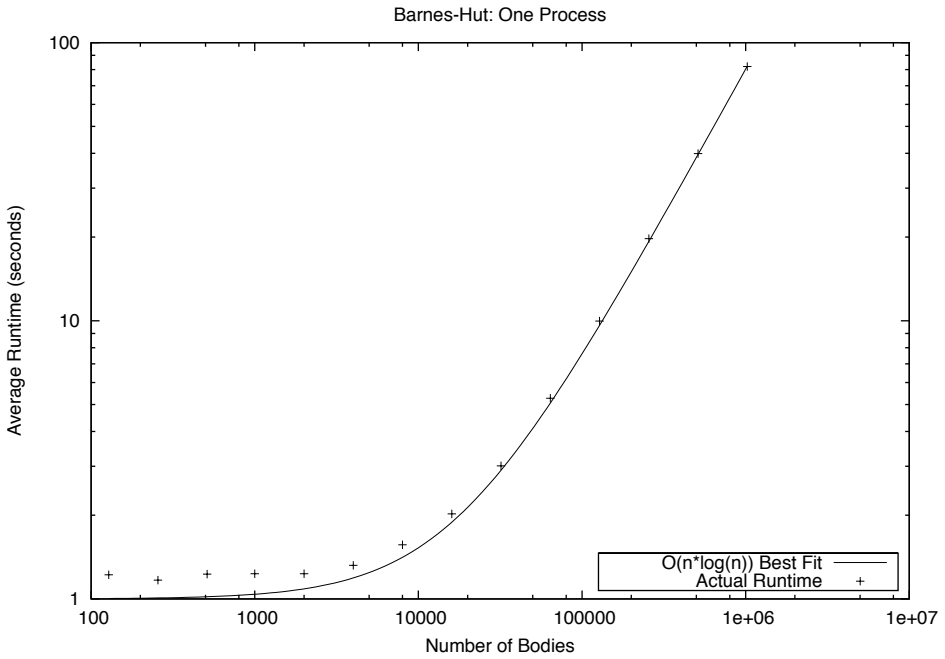

Figure 3.1: Barnes-Hut simulation for various numbers of bodies using one process.

lation. Due to discovered limitations of SPR we were only able to test our program using small numbers of bodies with a small number of processes. Included in our test results is timing data which shows the slow performance of RDMA GET requests.

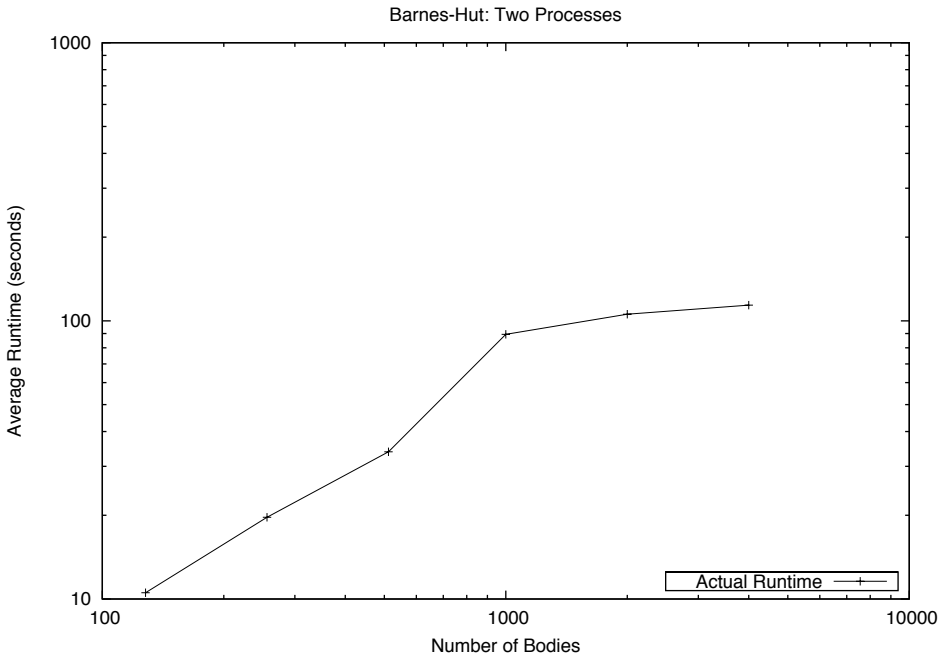

Figure 3.2: Barnes-Hut simulation for various numbers of bodies using two processes.

Number of Bodies	Total Number of GETs		Avg. Total GET Time		Avg. GET Time	
	Process 0	Process 1	Process 0	Process 1	Process 0	Process 1
128	450	316	15.3663	14.8517	0.0341	0.0469
256	598	689	18.0991	33.1700	0.0302	0.0481
512	905	1331	34.6184	43.6766	0.0382	0.0328
1000	1561	3199	79.4688	123.3075	0.0509	0.0385
2000	4276	3139	137.1376	207.9576	0.0320	0.0662

Table 3.1: Timing data for RDMA GETs done for Global Pointers during a Barnes-Hut Simulation. The times provided are in seconds.

CHAPTER 4

CONCLUSION

4.1 Discussion of Results

When running on a single process, our implementation seems to scale in $O(n * log(n))$ time as we increase the problem size. It does however run considerably slower than the Barnes-Hut implementation in [3]. We were unable to determine the cause of this slower performance.

Testing with two processes proved to be difficult as run times quickly grew with anything more than a few hundred bodies. Testing on process counts of anything above two proved to be equally as challenging as run times quickly escalated with even small body counts. We attribute this spike in simulation runtime to poor RDMA GET performance. The primary user of the `spr_get` function was the dereferencing of Global Pointers. As shown in Table 3.1 average GETs were far too slow to allow any sufficiently large simulation to take place.

SPR seems to scale decently on a single process but fails when adding additional processes. At a high level, we attribute this to the fact that the intra-process parallelism components of SPR are more mature than the inter-process parallelism components. However, our results indicate that there is still work to be done in improving performance for both.

4.2 Future Work

4.2.1 Parallelizing Tree Building

Currently, the process for the Build Octree phase of our parallel Barnes-Hut algorithm is executed completely in serial on each process. We did not

attempt to parallelize this section of the code because it represents a much smaller portion of the overall computation time used in our simulation. However, there may be significant gains to be had in parallelizing this portion of the code. Since the Octree building is recursive in nature, using Qthread's light-weight threading facilities may provide significant speedup for this particular phase of the simulation.

4.2.2 Increase RDMA GET Performance

The poor performance of our Barnes-Hut implementation on multiple processes was attributed to the slow performance of SPR's GET operation. We are unsure what the cause of this poor RDMA performance is. Work needs to be done to determine the root cause of this performance bottleneck. Possible causes could be hardware misconfiguration, software misconfiguration, or improper use of SPR.

4.2.3 Determine Cause of Slow Single-Process Performance

As mentioned in Section 4.1, while running on a single process our implementation runs slower than that in [3]. Work needs to be done in order to determine what performance bottlenecks are occurring that cause this slower performance. Possible causes could be cache misses, software misconfiguration, or improper use of SPR facilities.

4.2.4 Symmetric Malloc for Global Vector usage

When a Global Vector is constructed (and in several other cases), an all-to-all communication is required so that each process can exchange the pointer to it's value array with every other process. All-to-all communications are expensive and don't scale. We would like to implement something similar to the `shmalloc` function provided by SHMEM [15]. `shmalloc` allocates memory from a "symmetric heap". When memory is allocated from a symmetric heap, it is allocated at the same virtual memory address for each process. By allocating the value vector at the same virtual memory address for each process, every time a new Global Vector is constructed a pointer exchange

would no longer be necessary. This would also conserve memory, as value vector addresses for other processes do not need to be stored on every single process.

CHAPTER 5

REFERENCES

[1] J. Barnes and P. Hut, "A hierarchical 0 (n log iv) force-calculation algorithm," *nature*, vol. 324, p. 4, 1986.

[2] "Berkely upc." [Online]. Available: http://upc.lbl.edu

[3] J. Zhang, B. Behzad, and M. Snir, "Optimizing the barnes-hut algorithm in upc," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063485 pp. 75:1–75:11.

[4] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, 2008, pp. 1–8.

[5] D. Greenberg, R. Brightwell, L. Fisk, A. McCabe, and R. Riesen, "A system software architecture for high end computing," in *Supercomputing, ACM/IEEE 1997 Conference*, 1997, pp. 53–53.

[6] E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, pp. 789–828, 1996.

[7] P. Threads, "Posix. 1c," *Threads extensions (IEEE Std 1003.1 c-1995)*, vol. 35.

[8] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, IEEE, Ed., Pisa, Italie, Feb. 2010. [Online]. Available: http://hal.inria.fr/inria-00429889

[9] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "Methodological considerations and characterization of the splash-2 parallel application suite," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 24–36.

[10] R. W. Numrich and J. Reid, "Co-array fortran for parallel programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, Aug. 1998. [Online]. Available: http://doi.acm.org/10.1145/289918.289920

[11] J. P. Singh, W.-D. Weber, and A. Gupta, "Splash: Stanford parallel applications for shared-memory," *ACM SIGARCH Computer Architecture News*, vol. 20, no. 1, pp. 5–44, 1992.

[12] J. P. Singh, "Parallel hierarchical n-body methods and their implications for multiprocessors," 1993.

[13] "Threading building blocks." [Online]. Available: http://threadingbuildingblocks.org/

[14] "Taub campus cluster." [Online]. Available: https://campuscluster.illinois.edu/hardware/

[15] R. Barriuso and A. Knies, "Shmem users guide for c," Technical report, Cray Research Inc, Tech. Rep., 1994.

APPENDIX A

PPL SPECIFICATION

This is the specification for the PPL library as implemented by the program described in this paper.

A.1 Global pointers

Global pointers are pointers that can point to memory on remote nodes, and of course, they can point to local memory too. A global pointer contains two parts: a process id and an address on that process. It's internal structure is implementation dependent. Dereferencing a global pointer will return the data in the remote/local memory.

A.1.1 Definition

```
template <typename T> class gptr;
```

`T` is the type of the value `gptr` points to.

A.1.2 Member methods

`gptr()`
: Default constructor which creates a NULL Global Pointer.

`const T get_value()`
: Returns the value pointed at by the Global Pointer. This function is a synonym for the the `*` operator.

`const T get_value(ptrdiff_t i)`
: Return the value pointed at by this Global Pointer plus the offset `i`. This function is a synonym for the `[]` operator.

`void set_value(const T& x)`

Sets the value pointed at by this Global Pointer. This function is a synonym for `*gp = x`.

`void set_value(ptrdiff_t i, const T& x)`

Sets the value pointed at by this Global Pointer plus the offset `i`. This function is a synonym for the `gp[i] = x`.

`int process()`

Return rank of the process to which this Global Pointer has affinity.

`bool is_null()`

Returns whether or not the Global Pointer is null.

`void set_null()`

Sets the Global Pointer to null.

`bool is_local()`

Return true if the Global Pointer has affinity to the executing process. Returns false otherwise.

`operator Y*()`

Casts the Global Pointer to a local pointer of type `Y*`. This function is only legal when `gp.is_local()` returns true. If this function is called on a Global Pointer that does not have affinity to the current process, NULL is returned.

`operator gptr<Y>()`

Casts the Global Pointer to a Global Pointer of type `gptr<Y>`.

`const T operator*() const`

Returns the value pointed to by the Global Pointer. This operator can only be used as right value.

`const std::auto_ptr<T> operator->()`

May only be used as right value. This operator is best used when `T`'s size is small.

`gptr<T>& operator++()`

Increments the address pointed to by the Global Pointer.

`gptr<T>& operator--()`

Decrements the address pointed to by the Global Pointer.

`const gptr<T> operator++(int)`

Increments the Global Pointer by the amount specified.

`const gptr<T> operator--(int)`

Decrements the Global Pointer by the amount specified.

`const T operator[](ptrdiff_t i)`

Returns the ith value pointed to by the Global Pointer. This operator can only be used as right value non-member methods.

`gptr<T> operator + (const gptr<T>& a, ptrdiff_t i)`

Return a Global Pointer which points to the address at `a + i`.

`gptr<T> operator + (ptrdiff_t i, const gptr<T>& a)`

Return a Global Pointer which points to the address at `i + a`.

`bool operator == (const gptr<T>& a, const gptr<T>& b)`

Return true if `a` and `b` point to the same address. Returns false otherwise.

`bool operator != (const gptr<T>& a, const gptr<T>& b)`

Return true if `a` and `b` point to different addresses. Returns false otherwise.

`ptrdiff_t operator - (const gptr<T>& a, const gptr<T>& b)`

Returns the difference between `a` and `b`.

A.2 Global variables

Global variables are variables that can be referenced by all processes by their names. A feature of global variables is that once they are cached, they can be accessed just like local variables but without changing the names to access them. The common usage pattern of a global variable is (suppose `gx` a global variable):

```
if (myproc == pid) // One process writes gx
  gx =x;
ppl\_barrier();    // Make sure the write to gx is completed
gx.cache();        // Processes cache the value of gx locally
y = gx;            // Access the local copy without changing the name
```

A.2.1 Definition

```
template <typename T> class gvar;
```

`T` is type of the variable being shared.

A.2.2 Member methods

`gvar()`

Default constructor. Initialized to Global Variable to the default value for type `T`.

`gvar(const T& x)`

Construct a Global Variable with initial value `x`.

`gvar<T>& operator= (const gvar<T>& gy)`

Copy constructor. Assigns `gy` to this gvar, i.e., `gx = gy`. This constructor is collective.

`gvar<T>& operator= (const T& x)`

Assign a local variable `x` to this gvar, i.e., `gx = x`.

`void set(const T& x)`

Set value of this gvar. This function is synonymous with `gx = x`.

`void cache()`

Cache the value of the Global Variable locally.

`const T get()`

Return value of `gx`.

`operator T()`

Type conversion to a local type. This allows developers to write expressions like `x = gx`.

A.3 Global vectors

A global vector is a 1D co-array but can shrink or expand dynamically. The feature of a global vector is that it allows application developers to access remote elements, but when local elements the application developer has the same efficiency as accessing a local array.

A.3.1 Definition

```
template <typename T> class gvec;
```

`T` is the element type of the global vector.

A.3.2 Member methods

`gvec()`
: Construct an empty vector. This is a collective constructor.

`gvec(size_t size)`
: Construct a vector of length `size`. This is a collective constructor.

`gvec(size_t size, const T& x)`
: Construct a vector of length `size`, with each element being set to an intial value of `x`. This is a collective constructor.

`T& operator[](size_t i)`
: Return a reference to the i-th element in the vector on this process.

`const T get(size_t i, int proc)`
: Return the value of the i-th element on process `proc`.

`void set(size_t i, const T& val, int proc)`
: Set the value of the i-th element on process `proc` to be `val`.

`gptr<T> get_addr(size_t i, int proc)`
: Return a Global Pointer to the i-th element on process `proc`.

`void renew(size_t size)`
: Resize the vector to length `size` without preserving old data. This is a collective operation.

`void resize(size_t size)`

Resize the vector to length `size` while preserving old data.

`size_t size()`

Return size of the vector.

`void clear()`

Clear the vector. After the operation, the vector's size will be zero.

`template<typename Reducer> void all_reduce(Reducer op)`

Do a multi-element all reduction on vectors on all processes and overwrite the old value with the result. `op` is a user defined functor, such as `std::plus<uint64_t>()`

A.4 Global memory access

A.4.1 Blocking memory put/get

`void ppl_memput(gptr<void> dest, const void* src, size_t nbytes)`

Put `nbytes` bytes of data from local address `src` to remote memory with address `dest`.

`void ppl_memget(void *dest, const gptr<void> src, size_t nbytes)`

Get `nbytes` bytes of data from remote memory with address `src` and store it at local address `dest`.

A.4.2 Blocking multiple fix-sized items put/get

```
void ppl_memput_ilist(size_t dstcount,
                      gptr<void> dstlist[],
                      size_t dstlen,
                      size_t srccount,
                      void* const srclist[],
                      size_t srclen )
```

Scatter multiple items of the fixed size `srclen` bytes from local addresses `srclist[0],...,srclist[srccount-1]`, and store them to remote addresses `dstlist[0],...,dstlist[dstcount-1]`. Each remote address will be stored `dstlen` bytes. Precondition is `dstcount*dstlen = srccount*srclen`.

```
void ppl_memput_ilist(size_t dstcount,
                      gptr<void> dstlist[],
                      size_t dstlen,
                      size_t srccount,
                      void* const srclist[],
                      size_t srclen )
```

Scatter multiple items of the fixed size `srclen` bytes from local addresses `srclist[0],...,srclist[srccount-1]` and store them to remote addresses `dstlist[0]...dstlist[dstcount-1]` Each remote address will be stored `dstlen` bytes. Precondition is `dstcount*dstlen = srccount*srclen`.

A.4.3 Blocking multiple variable-sized items put/get

Blocking on multiple variable-sized items requires use of the following, PPL-defined data types.

```
typedef struct {
  void{*} addr;
  int len;
}ppl_lmemvec_t;

typdef struct {
  gptr<void> addr;
  int len;
} ppl_gmemvec_t;


void ppl_memput_vlist(size_t dstcount,
                      ppl_gmemvec_t const dstlist[],
                      size_t srccount,
                      ppl_lmemvec_t srclist [])
```

Scatter multiple items of the variable size from local addresses `srclist[0].addr,...,srclist[srccount-1].addr`. The item sizes are `srclist[0].len,...,srclist[srccount-1].len` respectively. The data is stored to the remote addresses `dstlist[0].addr,...,dstlist[dstcount-1].addr`. The destination addresses will be stored to `dstlist[0].len,...,dstlist[dstcount-1].len` bytes respectively. Precondition is `dstlist[0].len + ... + dstlist[dstcount-1].len` is equal to `srclist[0].len + ... + srclist[dstcount-1].len`.

```
void ppl_memget_vlist(size_t dstcount,
                      ppl_lmemvec_t const dstlist[],
                      size_t srccount,
                      ppl_gmemvec_t srclist [])
```

Gather multiple items of the variable size from remote addresses `srclist[0].addr,...,srclist[srccount-1].addr`. The item sizes are `srclist[0].len,...,srclist[srccount-1].len` respectively. The data is stored to the remote addresses `dstlist[0].addr,...,dstlist[dstcount-1].addr`. The destination addresses will be stored to `dstlist[0].len,...,dstlist[dstcount-1].len` bytes respectively. Precondition is `dstlist[0].len + ... + dstlist[dstcount-1].len` is equal to `srclist[0].len + ... + srclist[dstcount-1].len`.

A.4.4 Non-blocking memory put/get

```
ppl_handle_t ppl_memput_async(gptr<void> dest,
                              const void* src,
                              size_t nbytes )
```

Non-blocking version of `ppl_memput()`. The handle returned can be used to wait for the put to complete.

```
ppl_handle_t ppl_memget_async(void *dest,
                              const gptr<void> src,
                              size_t nbytes )
```

Non-blocking version of `ppl_memget()`. The handle returned can be used to wait for the get to complete.

A.4.5 Non-blocking multiple fix-sized items put/get

```
ppl_handle_t ppl_memput_ilist_async(size_t dstcount,
                                    gptr<void> dstlist[],
                                    size_t dstlen,
                                    size_t srccount,
                                    void* const srclist[],
                                    size_t srclen )
```

Non-blocking version of `ppl_memput_ilist()`. The returned handle can be used to wait for the list of puts to complete.

```
ppl_handle_t ppl_memget_ilist_async(size_t dstcount,
                                    void* dstlist[],
                                    size_t dstlen,
                                    size_t srccount,
                                    gptr<void> const srclist[],
                                    size_t srclen )
```

Non-blocking version of `ppl_memget_ilist()`. The returned handle can be used to wait for the list of gets to complete.

A.4.6 Non-blocking multiple variable-sized items put/get

```
void ppl_memput_vlist(size_t dstcount,
                      ppl_gmemvec_t const dstlist[],
                      size_t srccount,
                      ppl_lmemvec_t srclist [])
```

Non-blocking version of `ppl_memput_vlist()`. The returned handle can be used to wait for the list of puts to complete.

```
void ppl_memget_vlist(size_t dstcount,
                      ppl_lmemvec_t const dstlist[],
                      size_t srccount,
                      ppl_gmemvec_t srclist [])
```

Non-blocking version of `ppl_memget_vlist()`. The returned handle can be used to wait for the list of gets to complete.

A.4.7 Communication completion

`void ppl_waitsync(ppl_handle_t handle)`

Wait until the specified communication has completed.

`void ppl_waitsync_all(ppl_handle_t *handles, size_t numhandles)`

Wait until all the specified communications have completed.

A.5 Global memory allocation

`gptr<void> ppl_alloc(size_t nbytes)`

Allocate a block of global memory of `nbytes` bytes on the calling process. Return the global pointer to it.

`void ppl_free(gptr<void> gp)`

Free global memory pointed to by `gp`.

A.6 Atomic operations

`T ppl_atomic_fetchadd(gptr<T> gp, T inc)`

`T` is any integral type. Atomically add `inc` to the value pointed to by `gp`. Return the old value before addition.

A.7 Rank and size of processes

`int ppl_myprocess()`

Return a zero-based process index of the calling process.

`int ppl_nprocess()`

Return the total number of processes.

A.8 Barrier

`void ppl_barrier()`

Wait until all processes reach the barrier point.

A.9 Initialization and Finalization

`void ppl_init()`

This function must be called at the beginning of program execution. It initializes the PPL runtime facilities.

`void ppl_init()`

This function must be called at the end of program execution. It cleans up all the facilities setup by `ppl_init`.